1A. Numpy Pandas Tutorial

January 21, 2020

1 Numpy

Provides various utilities functions on top of N-dimensional array such statistical calculation (.mean(), .std()) and vectorized operation.

A numpy array is a N-dimensional array, all of the same type, and is indexed by nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

[1]:
```python
from numpy import *

a = arange(15).reshape(3, 5)
print (a)

print ("Type of a:", type(a))
print ("Shape of a:", a.shape)
print ("# dimensions:", a.ndim)
print ("datatype of elements in a:", a.dtype.name)
print ("# of elements:", a.size)

b = array([6,7,8])
print ("Type of b:", type(b))

print ("1st elem in b:", b[0]) # print first element in b
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
Type of a: <class 'numpy.ndarray'>
Shape of a: (3, 5)
# dimensions: 2
datatype of elements in a: int64
# of elements: 15
Type of b: <class 'numpy.ndarray'>
1st elem in b: 6
```

1.0.1 Array creation

Numpy also provides many functions to create arrays. You can also read more about array creation here : https://docs.scipy.org/doc/numpy/user/basics.creation.html#arrays-creation

[2]:
```
import numpy as np

a = np.zeros((2,2))   # Create an array of all zeros
print ("a:\n", a)

b = np.ones((1,2))    # Create an array of all ones
print ("b:\n", b)

c = np.full((2,2), 7)  # Create an array with a constant
print ("c:\n", c)

d = np.eye(2)         # Create a 2x2 identity matrix
print ("d:\n", d)

e = np.random.random((2,2))  # Create an array filled with random values
print ("e:\n", e)
```

```
a:
 [[0. 0.]
 [0. 0.]]
b:
 [[1. 1.]]
c:
 [[7 7]
 [7 7]]
d:
 [[1. 0.]
 [0. 1.]]
e:
 [[0.27541152 0.3505647 ]
 [0.28650962 0.55297334]]
```

1.0.2 Array indexing

Numpy offers several ways to index into arrays.

Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

[3]:
```
# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
```

```
# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

```
2
77
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array.

[4]:

```
# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #          [ 6]
                             #          [10]] (3, 1)"
```

```
[5 6 7 8] (4,)
[[5 6 7 8]] (1, 4)
[ 2  6 10] (3,)
[[ 2]
 [ 6]
 [10]] (3, 1)
```

1.0.3 Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

[5]:
```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
print ("matrix sum:\n", x + y)
print ("matrix sum:\n", add(x, y))

# Elementwise difference; both produce the array
print ("matrix difference:\n", x - y)
print ("matrix difference:\n", subtract(x, y))

# Elementwise product; both produce the array
print ("matrix prodcut:\n", x * y)
print ("matrix prodcut:\n", multiply(x, y))

# Elementwise division; both produce the array
print ("matrix division:\n", x / y)
print ("matrix division:\n", divide(x, y))

# Elementwise square root; produces the array
print ("square root:\n", sqrt(x))

# Elementwise product by scaler
print ("matrix by scaler:\n", x * 2)
```

```
matrix sum:
 [[ 6.  8.]
 [10. 12.]]
matrix sum:
 [[ 6.  8.]
 [10. 12.]]
matrix difference:
 [[-4. -4.]
 [-4. -4.]]
matrix difference:
 [[-4. -4.]
 [-4. -4.]]
matrix prodcut:
 [[ 5. 12.]
 [21. 32.]]
matrix prodcut:
 [[ 5. 12.]
 [21. 32.]]
```

```
matrix division:
 [[0.2        0.33333333]
 [0.42857143 0.5       ]]
matrix division:
 [[0.2        0.33333333]
 [0.42857143 0.5       ]]
square root:
 [[1.         1.41421356]
 [1.73205081 2.        ]]
matrix by scaler:
 [[2. 4.]
 [6. 8.]]
```

1.0.4 Linear algebra operations

[6]:
```
A = array( [[1,1], [0,1]] )
B = array( [[2,0], [3,4]] )

print ("A*B:\n", A*B) # elementwise product

print ("dot(A,B):\n", dot(A,B)) # matrix product

a = ones((2,3), dtype=int) # create matrix all 1s of size 2 x 3
b = random.random((2,3)) # create matrix of size 2 x 3, randomly

a *= 3 # multiply each element by 3
print ("a:\n", a)

b += a  # add two matrices, and assign to matrix b
print ("b\n:", b)
```

```
A*B:
 [[2 0]
 [0 4]]
dot(A,B):
 [[5 4]
 [3 4]]
a:
 [[3 3 3]
 [3 3 3]]
b
: [[3.53631989 3.56515941 3.81832469]
 [3.48555512 3.62293779 3.90788717]]
```

[7]:
```
a = random.random((2,3))

print ("matrix a:\n", a)
```

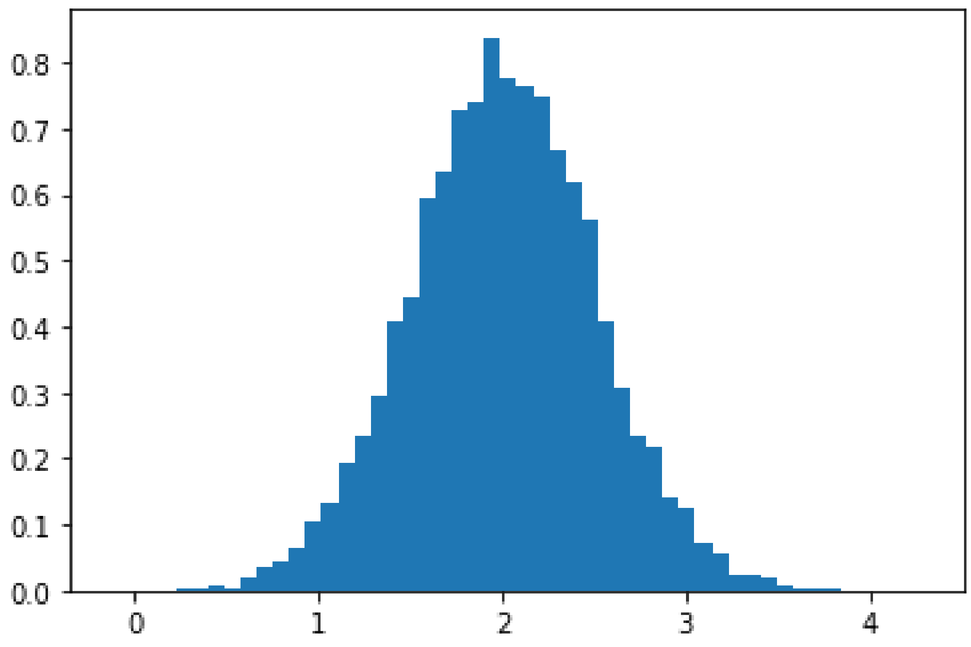
```python
print ("sum:\n", a.sum())
print ("min:\n",a.min())
print ("max:\n",a.max())
```

```
matrix a:
 [[0.1013491  0.82826631 0.66070218]
 [0.4172395  0.97820703 0.15785489]]
sum:
 3.143619006966839
min:
 0.10134909780790013
max:
 0.9782070292705993
```

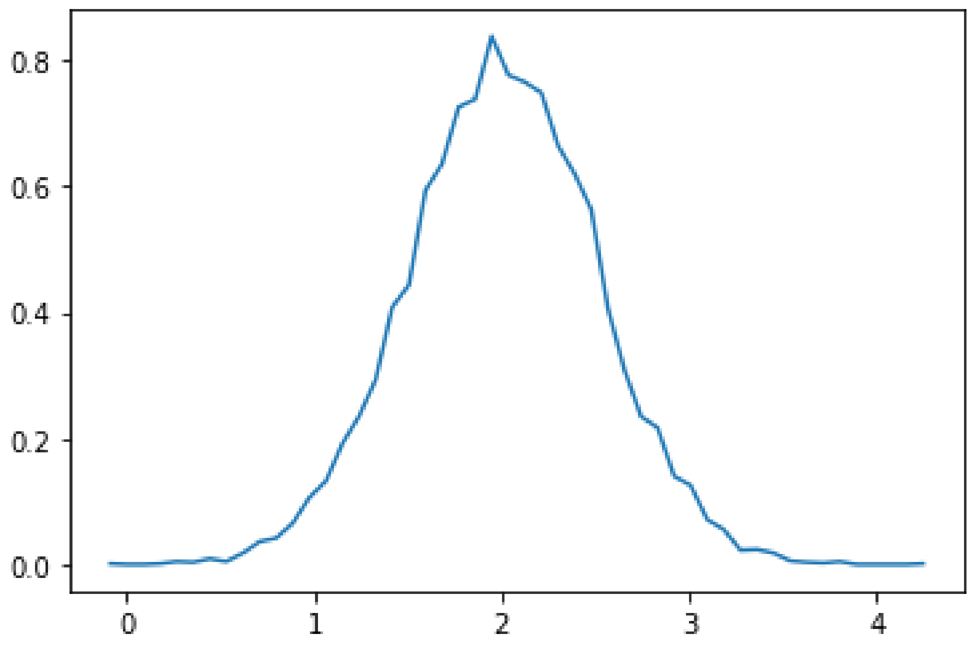
[8]:
```python
# show histogram of elements in array
import numpy
import pylab

# Build a vector of 10000 normal deviates with variance 0.5^2 and mean 2
mu, sigma = 2, 0.5
v = numpy.random.normal(mu,sigma,10000)
print ("v:", v)

# Plot a normalized histogram with 50 bins
pylab.hist(v, bins=50, normed=1)       # matplotlib version (plot)
pylab.show()

# Compute the histogram with numpy and then plot it
(n, bins) = numpy.histogram(v, bins=50, normed=True)  # NumPy version (no plot)
pylab.plot(.5*(bins[1:]+bins[:-1]), n)
pylab.show()
```

```
v: [1.07589726 1.96353657 1.51852197 … 3.52475627 2.24968662 1.32230166]
```

```
/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:11:
MatplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.1.
Use 'density' instead.
  # This is added back by InteractiveShellApp.init_path()
```



```
/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:15:
VisibleDeprecationWarning: Passing `normed=True` on non-uniform bins has always
been broken, and computes neither the probability density function nor the
probability mass function. The result is only correct if the bins are uniform,
when density=True will produce the same result anyway. The argument will be
removed in a future version of numpy.
  from ipykernel import kernelapp as app
```


0.0
0.2
0.4
0.6
0.8
0
1
2
3
4